LogRocket
Frontend Analytics

Rust microservices in server-side WebAssembly

October 25, 2022 · 6 min read

The Rust programming language has gained mainstream adoption in the past several years. It is consistently ranked as the most beloved programming language by developers and has been accepted into the Linux kernel. Rust enables developers to write correct and memory-safe programs that are as fast and as small as C programs. It is ideally suited for infrastructure software, including server-side applications, that require high reliability and performance.

However, for server-side applications, Rust also presents some challenges. Rust programs are compiled into native machine code, which is not portable and is unsafe in multi-tenancy cloud environments. We also lack tools to manage and orchestrate native applications in the cloud.

Hence, server-side Rust applications commonly run inside VMs or Linux containers, which bring significant memory and CPU overhead. This diminishes Rust's advantages in efficiency and makes it hard to deploy services in resource-constrained environments, such as edge data centers and edge clouds. The solution to this problem is WebAssembly (Wasm).

Started as a secure runtime inside web browsers, Wasm programs can be securely isolated in their own sandbox. With a new generation of Wasm runtimes, such as the Cloud Native Computing Foundation's WasmEdge Runtime, you can now run Wasm applications on the server. You can compile Rust programs to Wasm bytecode, and then deploy the Wasm application in the cloud.

According to a study published in IEEE Software, Wasm apps can be 100x faster (especially at startup) and 1/100 smaller compared to natively compiled Rust apps in Linux containers. This makes them especially well suited for resource-constrained environments such as edge clouds.

Wasm runtime sandboxes have much smaller attack surfaces and provide better isolation than Linux containers. Furthermore, Wasm runtime is portable across operating systems and hardware platforms. Once a Rust program is compiled into Wasm, it can run everywhere from development to production and from the cloud to the edge.

> *The anatomy of a Rust microservice in a WebAssembly sandbox.*

In this article, we'll cover the tools, libraries, APIs, frameworks, and techniques required to build microservices in Rust. We'll also demonstrate how to deploy, run, and scale these microservices in WasmEdge WebAssembly Runtime.

Jump ahead:

- Prerequisites
- Creating a web service
- Creating a web service client
- Creating a database client

- Building, running, and deploying the microservice
- Going to production

Prerequisites

To follow along with this article, you should have the following:

- Basic knowledge of the microservice design pattern
- Basic knowledge of the Linux operating system
- Familiarity with the Rust programming language
- Basic knowledge of SQL databases

Creating a web service

The microservice is foremost a web server. WasmEdge Runtime supports asynchronous and non-blocking network sockets. You can write networking applications in Rust, compile them into Wasm, and run them in the WasmEdge Runtime. In the Rust ecosystem, WasmEdge supports the following:

- The tokio and mio crates for asynchronous networking applications
- The hyper crate for HTTP server and client applications

The example below, from the microservice-rust-mysql demo app, shows how to create a web server in hyper for WasmEdge. The main listening loop of the web server is as follows:

```
    let addr = SocketAddr::from(([0, 0, 0, 0], 8080));
    let make_svc = make_service_fn(|_| {
        let pool = pool.clone();
        async move {
            Ok::<_, Infallible>(service_fn(move |req| {
                let pool = pool.clone();
                handle_request(req, pool)
            }))
        }
    });
    let server = Server::bind(&addr).serve(make_svc);
    if let Err(e) = server.await {
        eprintln!("server error: {}", e);
    }
    Ok(())
```

Once a request comes in, the event handler, `handle_request(),` is called asynchronously so that it can handle multiple concurrent requests. It generates a response based on the request method and path:

```
async fn handle_request(req: Request<Body>, pool: Pool) ->
Result<Response<Body>, anyhow::Error> {
    match (req.method(), req.uri().path()) {
        (&Method::GET, "/") => Ok(Response::new(Body::from(
            "... ...",
        ))),

        // Simply echo the body back to the client.
        (&Method::POST, "/echo") =>
Ok(Response::new(req.into_body())),

        (&Method::GET, "/init") => {
            let mut conn = pool.get_conn().await.unwrap();
            "DROP TABLE IF EXISTS orders;".ignore(&mut conn).await?;
            "CREATE TABLE orders (order_id INT, product_id INT,
quantity INT, amount FLOAT, shipping FLOAT, tax FLOAT,
shipping_address VARCHAR(20));".ignore(&mut conn).await?;
            drop(conn);
            Ok(Response::new(Body::from("{\"status\":true}")))
        }
```

Now we have an HTTP server for the web service.

Creating a web service client

A typical web service also needs to consume other web services. Through tokio and/or mio crates, WasmEdge applications can easily incorporate HTTP clients for web services. The following Rust crates are supported in WasmEdge:

- The reqwest crate for easy-to-use HTTP clients
- The http_req crate for HTTP and HTTPS clients

The following example shows how to make an HTTP POST against a web service API from a microservice:

```
    let client = reqwest::Client::new();

    let res = client
        .post("http://eu.httpbin.org/post")
        .body("msg=WasmEdge")
        .send()
        .await?;
    let body = res.text().await?;

    println!("POST: {}", body);
```

Creating a database client

Most microservices are backed by databases. The following Rust crates for MySQL drivers are supported in WasmEdge:

- The myql crate is a synchronous MySQL client
- The mysql_async is an asynchronous MySQL client

The example below shows how to insert a set of records into a database table:

```
    let orders = vec![
        Order::new(1, 12, 2, 56.0, 15.0, 2.0,
String::from("Mataderos 2312")),
        Order::new(2, 15, 3, 256.0, 30.0, 16.0, String::from("1234
NW Bobcat")),
        Order::new(3, 11, 5, 536.0, 50.0, 24.0, String::from("20
Havelock")),
        Order::new(4, 8, 8, 126.0, 20.0, 12.0, String::from("224
Pandan Loop")),
        Order::new(5, 24, 1, 46.0, 10.0, 2.0, String::from("No.10
Jalan Besar")),
    ];

    r"INSERT INTO orders (order_id, production_id, quantity, amount,
shipping, tax, shipping_address)
      VALUES (:order_id, :production_id, :quantity, :amount,
:shipping, :tax, :shipping_address)"
        .with(orders.iter().map(|order| {
            params! {
                "order_id" => order.order_id,
```

The below example shows how to query a database table and return a collection of records:

```
    let loaded_orders = "SELECT * FROM orders"
        .with(())
        .map(
            &mut conn,
            |(order_id, production_id, quantity, amount, shipping,
tax, shipping_address)| {
                Order::new(
                    order_id,
                    production_id,
                    quantity,
                    amount,
                    shipping,
                    tax,
                    shipping_address,
                )
            },
        )
        .await?;
    dbg!(loaded_orders.len());
    dbg!(loaded_orders);
```

Building, deploying, and running the microservice

The microservice-rust-mysql project provides a complete example of a database-driven microservice. Let's use it as an example to build, deploy, and run this service.

The Docker CLI and Docker Desktop provide seamless support for WasmEdge application development. From the root directory of the project repo, you just need a single command to build and bring up all the components of the microservice (i.e., the WasmEdge application and a MariaDB database server):

```
docker compose up
```

You can then test the CRUD operations on the database through the microservice using curl.

Alternatively, you can:

- Install the Rust compiler
- Install WasmEdge Runtime
- Install the MySQL database server

The commands below install the above prerequisites on a Linux system:

```
// Install Rust
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

// Install WasmEdge
curl -sSf
https://raw.githubusercontent.com/WasmEdge/WasmEdge/master/utils/insta
 | bash -s -- -e all

// Install MySQL. It is available as a package in most Linux distros
sudo apt-get update
sudo apt-get -y install mysql-server libmysqlclient-dev
sudo service mysql start
```

Next, build the microservice application into Wasm bytecode:

```
cargo build --target wasm32-wasi --release
```

Then, start the microservice in WasmEdge Runtime:

```
wasmedge --env
"DATABASE_URL=mysql://user:passwd@127.0.0.1:3306/mysql"
order_demo_service.wasm
```

You can then use the microservice's web API to access the database:

```
// Init the database table
curl http://localhost:8080/init

// Insert a set of records
curl http://localhost:8080/create_orders -X POST -d @orders.json

// Query the records from the database
curl http://localhost:8080/orders

// Update the records
curl http://localhost:8080/update_order -X POST -d
@update_order.json

// Delete a record by its id
curl http://localhost:8080/delete_order?id=2
```

Going to production

So far, we've seen a complete database-driven microservice in action. In the real world, however, a company could have hundreds of microservices. They must be managed and orchestrated by cloud-native frameworks such as Kubernetes.

WasmEdge applications are fully OCI-compliant. They can be managed and stored in the Docker Hub or other Open Container Initiative repositories. Through crun integration, WasmEdge can run side-by-side with Linux

container applications in the same Kubernetes cluster. This repository showcases how to run WasmEdge applications in popular container toolchains including CRI-O, containerd, Kubernetes, Kind, OpenYurt, KubeEdge, and more.

WebAssembly runtime in the Kubernetes stack.

Furthermore, microservices are often deployed together with service frameworks. For example, Dapr is a popular runtime framework for microservices. It provides a "sidecar" service for each microservice. The microservice accesses the sidecar via the Dapr API to discover and invoke other services on the network, manage state data, and access message queues.

The Dapr SDK for WASI (WebAssembly System Interface) enables WasmEdge-based microservices to access their attached Dapr sidecars. A complete demo application with a Jamstack static web frontend, three microservices, and a database service is also available.

Conclusion

In this article, we discussed why WebAssembly is a great runtime sandbox format for Rust-based server-side applications. We walked through concrete code examples demonstrating how to create an HTTP web service, consume other web services, access relational databases from Rust, and then run the compiled applications in WasmEdge. We also investigated deployment concerns, such as Kubernetes and Dapr integrations.

With those crates and template apps, you will be able to build your own lightweight microservices in Rust!

Besides microservices, the WasmEdge Runtime can be widely used as an application sandbox in many different scenarios:

- The flows.network is a serverless platform for SaaS automation. You can create bots, customizations, and connectors for SaaS products using Rust and WebAssembly
- You can create User Defined Functions (UDFs) and Extract-Transform-Load (ETL) functions for databases in WebAssembly, and have those functions embedded or colocated with the database in a "serverless" fashion
- You can create portable and high-performance applications for edge devices, running Android, Open Harmony, and even the seL4 RTOS, using WebAssembly
- WebAssembly is widely used as a runtime for blockchains to execute smart contracts. WasmEdge, for instance, runs nodes and smart contracts for Polkadot, FileCoin, and XRP (Ripple) networks. The next-generation Ethereum blockchain VM, known as the ewasm, is also based on WebAssembly

To learn more about WasmEdge, see the official docs.

LogRocket: Full visibility into web frontends for Rust apps

Debugging Rust applications can be difficult, especially when users experience issues that are difficult to reproduce. If you're interested in monitoring and tracking performance of your Rust apps, automatically surfacing errors, and tracking slow network requests and load time, try LogRocket.

Michael Yuan Follow

Michael Yuan is the maintainer of WasmEdge Runtime, a cloud native WebAssembly sandbox project under CNCF. He is the author of six books on software engineering. Connect with Michael on Twitter or GitHub.

#rust #webassembly

4 Replies to “Rust microservices in server-side WebAssembly”

Zoraiz Says:
October 31, 2022 at 4:01 pm

Reply

Great article. However, is there a need to go so far as porting rust apps to wasm just to reliably run them in the cloud environment? Are there not existing technologies that are built and optimized for that environment, or are there specific advantages to replacing them with rust?

Michael Yuan Says:
October 31, 2022 at 5:26 pm

Reply

Hi, you will need to compile your Rust app to Wasm — not rewriting them in another API. So, why compile to Wasm instead of x86 and arm64? That is because the Wasm “container” is safer, faster and lighter than VMs / LXCs required to run x86/arm64 apps in the cloud.

John Says:
February 13, 2023 at 3:03 pm

Reply

Native rust is still faster than wasm, so what are the other advantages beside container size which is like nothing with nowadays capabilities. Nobody cares if it’s 3 mb or 50 anymore, memory is cheap

Michael Yuan Says:
February 14, 2023 at 1:18 pm

Reply

Native Rust means no container at all. You cannot run “native” in a cloud environment. You need either a container or a VM.

In a typical service mesh today, over 50% of the computing resources (CPU, disk, memory) are used on container overhead. So, I believe the weight, speed, portability, and security of the container are some of the most important issues in cloud computing today.

Leave a Reply